Project Final Report

Game Name: **Spy City**

Development Team
Ryan Gallagher
Robert Lippert

Overview:

Concept and Goals:

Our goal was to emulate the driving part of GTA3. Our concept was to do this by loading the GTA3 model and level files but providing our own engine and physics, etc.

Features:

- Single Player mode
- Converter to load GTA3 provided level/model files
- Droppable quaternion-based 3rd person camera
- Car model with see-through windows and semi-realistic physics
- Collision detection with entire polygon-based world
- Octree-based visibility occlusion for playable frame-rate
- LOD for building models
- In-game TCL-based console for on-the-fly manipulation of all game state
- Skydome that renders two independently animated textures and simulates daytime-nighttime changes.

Controls:

- Directional keys for car movement
- <SPACE> for handbrake
- <C> to toggle dropping the camera/placing behind car
- <ESC> to toggle the console
- <V> to toggle collision debug lines
- <ALT-ENTER> to enter/exit full-screen mode
- When camera is dropped (not behind car):
  - <LMB> and drag to pan the camera
  - <RMB> and drag to orbit the camera
  - <MMB> and drag to dolly the camera

Development Summary:

Code:

Our code is split between a number of scripts that define the game and a whole lot of C++ code that forms the engine. In an attempt at a data-driven design, all of our data is located in the data\ directory including scripts, models, meshes, sounds, etc. The rest of the C++ code for the engine is in the code\ directory.

Its hard to list exactly the code that we wrote because a lot of our c++ code was modifications to existing nebula classes especially for the collision detection and physics part. All of our code to control the car and to handle the octree was cleanly separated

into a "ngame" module but the only way to tell what other code was written is to look at the CVS logs and see what is modified. There were also a LOT of scripts including in-game TCL to create all the objects needed (data-driven) and a bunch of perl scripts and one C++-program (dff2n3d) to handle converting the GTA3 level and model files to a format usable by Nebula.

We used a lot of external libraries and such including the Nebula engine (www.sourceforge.net/projects/nebuladevice) which formed a nice backbone for the engine and the ODE physics library (www.q12.org/ode/ode.html) for physics (which didn't turn out that great). By using the Nebula engine we were able to focus our efforts on model-level techniques to speed up rendering (ie LOD/octree) instead of having to worry about dealing with triangles and other lower-level issues. It also makes the code much more extensible as adding effects like sound-support and particle effects would take very little time.

Content:

Most of the content comes from GTA3 and was converted to a format usable by nebula using a C++ program we wrote. It properly converts all geometry and lighting/color/texturing data to insure the models look just like in GTA3. All the textures also come from GTA3 and were hand-converted using a freely available program. The car model was found somewhere on the internet (freely available) and the remaining textures (ground plane, etc.) are simple public domain textures. There was practically no original content created as neither of us are 3d artists.

Key Technical Challenges:

- Converting the GTA3 files to a format usable by Nebula
  - The GTA3 model files for the buildings and the roads, etc. are all stored in a proprietary format to which the specification is not publicly available. Some hobbyists have made an effort to document the format and from that information we attempted to write a converter. Unfortunately the information available was only applicable to some of the models and caused our program to not work on others. Thus we had to do a lot of looking at the files with a hex editor to understand how the data was actually stored before we could get all models converted correctly, which took much more time than we had allotted (because we assumed we could simply follow the available information).
- Implementing realistic physics
  - Our proposal called for using the ODE physics library because we believed it to be a very fast and accurate rigid-body simulation. Unfortunately the method of simulation used by ODE (fixed-size timesteps) does not work for wheels that are rotating at high speeds (like our car) because they rotate more than one full revolution per time-step. ODE has a lot of parameters that are settable in order to fix problems like this but we were never able to find a collection of parameters that worked well and was still playable (doing so is like solving an equation in 15 variables).
- Nebula/ODE learning curve

- While both of us had looked at the nebula engine before the project (basically running the demos and skimming the docs), neither of us had actually done any programming with it. Thus during the course of the project we discovered a bunch of little issues that caused us to spend lots of time debugging and looking at their source code. Also, integrating the ODE physics and collision detection was a non-trivial task and while their was some source code available to do it was very buggy. Another problem we ran into was the massive number of models and textures that we were loading were just not planned for by the library creators. This caused all of the fixed size arrays in nebula and ODE to choke and required a number of changes of constants to get things to work correctly

Postmortem:

Things that went Right:

- GTA Model loader
  - Although it was a pain to do, we did end up able to load all of the models (a few problems remain but they're minor) and it made the city just Look Good, much better than the randomized city we had planned at the beginning.
- Car model with transparent windows
  - This took a lot of fiddling around with Deep Exploration on our model that we found, but wound up looking very good with wheels that actually turn correctly, transparent windows and everything we could think of.
- Octree/LOD visibility
  - Integrating the Octree with LOD is the only thing that gives us a playable framerate. Luckily since nebula already included a octree-class (although it is unused by default) all we had to do was integrate it into the render loop and check distances against our LOD thresholds. Very clean and easy and it yielded good performance.

Things that went Wrong:

- Collision
  - GTA3 included collision information along with the actual models which listed boxes and triangles to test against. However the triangles in the collision files were far too large to be handled by our triangles collision system (each wheel needs to be touching a different triangle for it to work right) and caused endless headaches trying to get it to work before we finally gave up. In the end we just use the model triangles for collision although since there are more triangles the performance is much worse. Another problem with collision we had was that the bounding box for the car body prevented the car from going up any steep slopes (it is a low-rider car =)) so as a quick hack we just decided to get rid of it and only test collisions against the wheels. Of course the problem with this is that when the car flips over the wheels still contact the ground, but this is a minor difficulty.
- Sound

- Sound support in nebula is supposed to be as easy as associating a .wav file with a particular 3d node in space and setting the listener position at the camera. But after trying many different configurations all we could manage to get was a very fuzzy sounding thing coming out of the speakers.

- Camera
  - The camera in the game is very simple because it has no physics associated with it at all and it does not collide with the world geometry. We had planned to implement a physically-based spring camera using the ODE physics system already in place but due to time constraints and our already bad experiences with ODE we gave up and stuck with what we knew worked.

Things learned:

- Replicating a commercial game engine is not an easy task
- Using off the shelf libraries is not always easier than writing your own code
- Building the engine in Release mode is 3-4X faster than Debug mode (very cool!)
- Adding in lots of debug visualizations is useful to detect when things go wrong
- Dealing with large numbers of models is not an easy task when it takes 5 minutes just to load the game in order to start debugging
- Its easy to concentrate on the technical features and forget about the gameplay
- There is always more to tweak